Improving Inference Privacy for Large Language Models using Fully Homomorphic Encryption

Rohit Mittal

Electrical Engineering and Computer Sciences
University of California, Berkeley

Technical Report No. UCB/EECS-2024-225
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2024/EECS-2024-225.html

December 19, 2024

Copyright © 2024, by the author(s).
All rights reserved.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission.

Acknowledgement

I would like to thank Professor Dawn Song for giving me the opportunity to work on this project, as well as my mentor Xiaoyuan Liu for his mentorship and guidance for the past two years both in my undergraduate and graduate career. I am truly grateful to have honed my skills under them and for their support throughout the process. I would also like to thank the Sunblaze lab and Tianneng Shi for enabling my access to their compute resources, without which this project would not have been possible. Lastly, I would like to thank my family, whose guidance, encouragement, and unwavering love have shaped me into the person I am today.

Improving Inference Privacy for Large Language Models using Fully Homomorphic Encryption

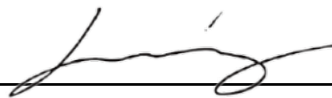
by Rohit Mittal

Research Project

Submitted to the Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, in partial satisfaction of the requirements for the degree of **Master of Science, Plan II**.

Approval for the Report and Comprehensive Examination:

Committee:



Professor Dawn Song
Research Advisor

Dec 19, 2024

(Date)

* * * * * * *

Professor Raluca Ada Popa
Second Reader

Dec 19, 2024

(Date)

Abstract

Improving Inference Privacy for Large Language Models using Fully Homomorphic Encryption

by

Rohit Mittal

Master of Science in Electrical Engineering and Computer Science

University of California, Berkeley

As large language models (LLMs) become more and more prevalent in our lives, concerns surrounding the privacy of their input and output data have been brought to the forefront of the debate around the use of this technology. In order for large language models to realize their full potential, solutions must be designed that protect the sensitive data in user queries. This report explores a solution to private inference through fully homomorphic encryption. The report showcases how fully homomorphic encryption (FHE) can be used to design a model that offloads intensive query computation to a remote server, similar to present-day LLM client-server models, while encrypting the query such that the server can compute over it while being oblivious to the plaintext query itself, ensuring privacy by design without having to trust the server. The report also discusses methods to increase the speed of the computation without directly revealing the inference result and examines the impacts of these methods on a working implementation of the Meta Llama 2 LLM where intensive query computations are offloaded to a server while still ensuring the privacy of the inputs and outputs.

Contents

Contents **i**

List of Figures **ii**

1 Introduction **1**

2 Background **3**
2.1 Large Language Models 3
2.2 Fully Homomorphic Encryption 4

3 Motivations **9**
3.1 Performance 9

4 Problem Definition **10**
4.1 Threat Model 10

5 Methodology **12**
5.1 FHE Scheme Selection 12
5.2 Quantization 13
5.3 Operation Selection 14
5.4 Final Model 15

6 Results **17**
6.1 Memory Usage 18
6.2 Bandwidth Usage 18
6.3 Token Inference Speed 19
6.4 Input Size Impact 20
6.5 Quantization Impact 20
6.6 Parallelization Impact 22
6.7 Conclusions 23
6.8 Future Work 24

Bibliography **26**

List of Figures

5.1 A diagram of the hybrid client-server model 16

6.1 Bandwidth per token on a five token output from a seven token input 18
6.2 Time per token on a five token output from a seven token input. 19
6.3 Total time spent on each FHE operator 20
6.4 Total time and bandwidth spent on a five token output based on the size of the input, scaling from one to seven tokens 21
6.5 Total time and bandwidth spent on a five token output from a seven token input based on the quantization of the model 22
6.6 Total load and inference time spent on a five token output from a seven token input based on the number of processors assigned to the server 23

Acknowledgments

I would like to thank Professor Dawn Song for giving me the opportunity to work on this project, as well as my mentor Xiaoyuan Liu for his mentorship and guidance for the past two years both in my undergraduate and graduate career. I am truly grateful to have honed my skills under them and for their support throughout the process. I would also like to thank the Sunblaze lab and Tianneng Shi for enabling my access to their compute resources, without which this project would not have been possible. Lastly, I would like to thank my family, whose guidance, encouragement, and unwavering love have shaped me into the person I am today.

Chapter 1

Introduction

When designing products for the use of individuals, businesses, and governments, it is vital for engineers to create a system that users can trust. This is especially true in the realm of products that incorporate large language models (LLMs). The large language model is a recent innovation that brings many novel product ideas to life, creating new opportunities to save time, automate tasks, and derive new insights from data that would not have been possible with older computation paradigms. However, in order for LLMs to reach their full potential, products have to be designed so that users are assured of the confidentiality of their data.

Privacy is a paramount concern in an increasingly digital world. The past decade has introduced the world to the value of big data, and with it, the responsibilities one must have for their own data. Users must now take into account the risk that data they provide to a product can be used in ways they may not condone, or that it may be leaked to malicious adversaries. Data breaches have a very tangible cost associated with them, with the legal fees for the Target data breach in 2013 exceeding $150,000 alone[5]. In addition, they can have additional costs both for the company and the overall economy when taking into account the loss of customer trust and the increased risk of identity theft from adversaries gaining access to private customer data. Privacy concerns have a measurable effect on the success of a product, with studies[10] showing that users change their behavior and interact less with products that they associate with breaches of privacy. Thus, ensuring that users can trust a product is an important factor in making sure the product can succeed.

Large language models face unique challenges in assuring user trust. Their design strongly depends on a user voluntarily providing data that may be sensitive, meaning that a user must have very high confidence in the privacy of their data. This is especially true for institutional use cases, where products incorporating LLMs may have to adhere to strict regulations on handling user data in order to follow government or company policies. One of the risk factors affecting the applicability of LLMs is the privacy concerns around data fed to it both in training and in deployment[23]. While there are some privacy preserving solutions for LLM training, there is still work to do in order to assure the privacy of user provided data in the query. This is because LLMs by their nature are highly computationally intensive,

meaning that in many use cases, users have to send their queries to remote servers with purpose-built hardware. This client-server model bypasses the need for users to carry their own hardware but introduces privacy concerns, as users must trust that the server they send their data to is not compromised. In addition, the user must trust the server provider and the LLM host to not leak or misuse their queries. This makes LLM products unsuitable for scenarios where regulators or policy makers may require sensitive data not to be sent to remote servers, limiting the use of LLMs in fields like health, finance, and other domains that use sensitive data.

There has been some work in using cryptography to ensure the privacy of the query in a client-server inference model. One example would be BOLT[22], which uses a combination of fully homomorphic encryption and secure multiparty computation (MPC) in order to create an inference protocol where only the client can see the results of the inference. There have been various iterations of using MPC for inference privacy, such as IRON[9] (BOLT's predecessor), and SecFormer[18]. Among the best performing contemporary MPC augmented models is Bumblebee[17], which claims to generate a token every 14 minutes on average when evaluating the LLaMA-7B model. However, these works only guarantee privacy against a semi-honest adversary which controls one party in the multi-party computation but strictly adheres to the protocol while trying to glean private information. These works do not guarantee privacy against a malicious adversary that does not need to follow the protocol, limiting their applicability in situations where an adversary can corrupt the protocol itself to learn private information.

In order to create a solution that works against a malicious adversary, this paper focuses on exploring a solution to this challenge that involves incorporating fully homomorphic encryption (FHE) into LLMs. FHE is a type of encryption that allows computations to be performed directly on encrypted ciphertexts while remaining oblivious to the decrypted plaintext. By incorporating FHE into LLMs, users can ensure that their encrypted queries are private by design. When the user sends an encrypted query to the server, the server can compute over it in FHE, but only the user has the ability to view the plaintext query and result. Unlike previous purely FHE solutions such as NEXUS[30], which performs an end-to-end FHE evaluation of a model, this paper will explore the idea of only offloading specific operations to be done in FHE while preserving privacy by doing low intensity operations on the client. This paper will explore the various possible methods of integrating these FHE operations, as well as discuss some of its challenges, such as choosing an FHE scheme, handling quantization, and choosing operations to perform in FHE. The paper will then conclude with an analysis of the culmination of this work: An implementation of the Meta LLaMA 2 LLM which offloads intensive query computation to a server while ensuring the privacy of the query through FHE.

Chapter 2

Background

2.1 Large Language Models

Large Language Models[21] are highly generalized artificial intelligence systems that can interpret and generate natural language. There are many different LLMs today, with some of the most popular including OpenAI's ChatGPT, Google's Gemini, and Facebook's LLaMA series of models. While still a relatively recent development in technology, they have quickly found uses as chatbots, translators, interpreters, and classifiers with versatility that was not possible with previous technology.

While the models differ in implementation and training, they all follow a core set of principles. Inputs to the models are tokenized into vectors of tokens, which can represent input characters or words depending on the implementation. LLMs take the tokenized vector and weigh each token by its perceived importance in order to emphasize the most relevant tokens. Normalization functions are used to stabilize inputs and make them converge more quickly. Activation functions fit their inputs to a non-linear function in order to assist the model in representing non-linear relationships[11]. All of these are vital components to the Transformer based architecture[26], and make up the layers that process the tokenized inputs. These layers involve combining vectors of input and intermediate products from the previous layers with pre-trained vectors to obtain the next layer's output. These operations generally involve linear transformations and other matrix-vector operations, which are very computationally intensive.

As such, most LLM computation is done using specialized hardware in the form of GPUs and TPUs. These allow for efficient and rapid parallel computation over large matrices and vectors which would otherwise take a CPU a long time to do sequentially. However, many applications of LLMs occur in environments without GPUs or TPUs, which limits their applicability. To resolve this, LLMs can be run on remote servers where the client's query is sent to a server with the required hardware. The server then quickly and efficiently produces a response with its stronger dedicated hardware. These servers are generally hosted by large scale cloud providers and can be orchestrated and accessed by many parties, causing concerns

for data privacy.

Work on privacy preservation in LLMs has led to some potential solutions in this space, each with some trade-offs. The conceptually simplest solution is to remove the server from the equation altogether. Local LLM computation can be done with many open source and open weight models but generally requires the user to have access to the specialized hardware needed to run the LLM on site. Server providers themselves can also provide some measures of security to query data, such as running servers in secure data centers and communicating with the client using secure protocols. However, these safeguards still require the user to trust the LLM provider, as the server itself can still see the query in plaintext. In order to preserve the secrecy of the plaintext from the provider, servers can integrate secure hardware enclaves that the LLM runs inside, such as the Keystone[14] secure enclave. This allows for remote attestation of the LLM's integrity and hides the plaintext query from the server itself, but requires the user to still trust the company and supply chain behind the secure hardware, as the enclave is the root of trust for the computation.

Secure multi-party computation[16] can be used to create a system where a group of non-colluding servers can compute over the query in a distributed fashion and no one server has access to the plaintext itself. Bumblebee[17] and BOLT [22] use multi-party computations in order to evaluate approximations of nonlinear functions, such as GELU and SoftMax. From a performance standpoint, MPC is well suited for these kinds of privacy-preserving computations and is able to do them with minimal communication overhead. However, as explored in BOLT[22], MPC has significant communication costs that make evaluating linear layers less performant. In addition, current works using MPC for secure inference only explore a threat model with a semi-honest adversary. This makes MPC for secure inference impractical in applications that require privacy guarantees against an adversary that is not restricted to semi-honesty.

2.2 Fully Homomorphic Encryption

Overview

Fully Homomorphic Encryption[19] is a special kind of encryption that allows for manipulation of the encrypted data without needing to decrypt it. At the highest level, FHE works by creating an encryption scheme that is compatible with addition and multiplication and is able to extrapolate those into mathematical and boolean operations.

There are many different schemes, but the ones this project focuses on are schemes that work with numerical inputs and outputs. BFV[27] and BGV[2] are schemes that use ring learning with errors (RLWE) to do integer and fixed-point arithmetic. To do this, the plaintext is encrypted with a public key and some random noise. The encryption process creates a ciphertext that can be represented as a first-degree polynomial. The polynomial is chosen such that it will return the plaintext when given a specific input known only to the client, also known as the secret key. Adding two ciphertexts together is as simple as adding

together the coefficients of each polynomial. If one has two ciphertexts in the form of $ax + b$ and $cx + d$, the addition of the two would be $(a + c)x + b + d$. When this is given back to the secret key holder, they can compute the input by substituting the value of x for the secret key. Similarly, multiplying the above ciphertexts involves creating a second-degree polynomial $(ac)x^2 + (bc + ad)x + bd.$. However, this creates a second-degree polynomial, for which the existing homomorphic operations cannot be done on. While addition and multiplication could be devised to work on second-degree polynomials as well, this process does not scale very efficiently and can cause loss of performance. To prevent this, rounds of communication with the secret key holder can be used to relinearize the polynomial to only have one degree. BFV and BGV differ in their approaches to relinearization, and it is considered the critical bottleneck for homomorphic operations. Coefficients for ciphertexts are chosen to be congruent to some modulus chosen at the beginning of the scheme in order to ensure the security of the scheme. In addition to this, the noise introduced when creating the ciphertexts can accumulate with homomorphic operations. To handle this noise, coefficients with noise can be shrunk by switching the modulus, both reducing the accumulated noise and shrinking the coefficients of the polynomial, making future operations more performant.

One limitation of integer based FHE is that it cannot be used directly for floating point arithmetic. An alternative scheme is CKKS[3], which absorbs the error generated by FHE operations into the error inherent in floating point operations. CKKS is approximate arithmetic, meaning that some error will be generated in the result, but the scheme can be tuned to have a larger or smaller modulus based on the need for performance or higher accuracy. The relinearization operation for CKKS also consumes one of the polynomial moduli from the ciphertext to relinearize the ciphertext. This introduces the concept of multiplicative depth to CKKS. Because the number of linearization operations is depending on the number of moduli, a given ciphertext can only do a certain amount of multiplication operations before exhausting its moduli.

One other important scheme for this project is TFHE[4]. TFHE is a high performance integer based FHE scheme which leverages a torus shaped algebraic structure for its ciphertexts. It aims to increase performance by making relinearization operations very performant and has the ability to increase performance further by being able to do certain operations simultaneously alongside relinearization for free through an operation called programmable bootstrapping.

Tooling

There are a number of software implementations and frameworks for FHE, spanning from high level APIs that abstract away the complexities of individual operations to low level primitives that the user can use to build up more complex cryptographic operations. One example is the Simple Encrypted Arithmetic Library from Microsoft[24]. SEAL provides a set of libraries that implement an API for FHE operations. It has implementations of the BFV, BGV, and CKKS encryption schemes and provides addition and multiplication as well as a host of more complex vector operations, such as vector addition, selection, and

rotation. SEAL itself is implemented in C++ but has ports to various other programming languages. One such port is the TenSEAL library[1], which allows for SEAL operations to be performed in Python through vectors that are based on the tensors used by the Tensorflow and Pytorch libraries. TenSEAL includes some useful tensor operations such as matrix-vector and matrix-matrix multiplication both in plaintext and ciphertext. Both SEAL and TenSEAL still expose low level implementation details to the user. For CKKS, this means the size of the polynomial modulus and the number of moduli are set by the user, and the user must have a deep understanding of the underlying scheme to utilize the libraries.

Another FHE implementation is the HEaaN homomorphic encryption library[12], which is a CKKS library implemented in C++ which can utilize general computing accelerators in the CPU such as the AVX-512 extension on Intel processors, or even utilize GPUs to accelerate homomorphic computation.

A lower level example of an FHE implementation is HElib[8], which implements BGV and CKKS in C++. The goal of HELib is to provide assembly-like low level functions for working with homomorphic ciphertexts, such as set, add, multiply, and shift. It also supports automatic bootstrapping and multithreading for improved results. Operators are also overloaded to work with plaintext objects as well to maximize interoperability.

Lattigo[13] is an FHE library implemented in Go that focuses on creating encrypted distributed systems. The library allows for the creation of lattice based circuits that can integrate both multiparty and homomorphic encryption together, with support for CKKS and BFV/BGV. The implementation flexibility allows for significant performance gains and circuits with a lot of capabilities.

Concrete ML

An implementation of interest to this project is the Concrete ML library[28]. Concrete ML is an implementation of the TFHE scheme in Python. It is designed from the ground up to be used in production FHE deployments that involve a client-server architecture for machine learning applications. As such, it contains integrations with many popular Python libraries, such as Numpy, Scikit-Learn, and Pytorch. Unlike SEAL, it abstracts away most of the complexity of TFHE and provides the user with a workflow that is easier to integrate into existing systems.

In order to create an FHE circuit using Concrete ML, a model is first compiled into Concrete. Concrete[29] is the underlying library behind Concrete ML which implements and runs FHE circuits. Compilation involves taking a set of FHE operations and converting them into an intermediate representation in Concrete that encompasses an FHE circuit. This compiled model can then be simulated in plaintext in order to judge the accuracy of the model without having to run it in FHE, allowing for faster development cycles. Inference can then be performed on the compiled model by calling it with the input, very similarly to how one would call a plain Torch model. During inference, keys are generated using a random number generator to encrypt the ciphertext, and then the ciphertext is operated on in FHE before the result is decrypted. This operation can be easily split into a client-

server model because Concrete ML also includes serialization of the compiled models and ciphertexts. This allows for a flow where a client can send a server encrypted input alongside a compiled model, and then the server can compute over the data in FHE before sending the computed ciphertext back to the client to decrypt, without the server ever having access to the unencrypted plaintext or the secret key.

This client-server approach allows for Concrete ML to create a circuit where operations are split between the client and the server. The `HybridFHEModel` compiler is able to take a Pytorch model along with a list of submodule names to compile and return a Pytorch module where the listed submodules are replaced by a remote call to a server. This call homomorphically generates keys, encrypts its inputs with those keys, and sends the ciphertext along with the compiled and serialized model to a server that implements the `HybridFHEServer`. The server then de-serializes the model and homomorphically computes the input over the provided model before sending the output back to the client. However, the server would not be able to derive any useful information about the inputs or the outputs from the ciphertexts due to the nature of FHE. It would have access to the ciphertext and would be able to modify it, but because the server is not in possession of the keys used to encrypt the ciphertext, it cannot access the plaintext inputs or outputs.

Concrete ML supports a great number of FHE operations out of the box. It supports in-place FHE compilation for a number of Pytorch modules, from arithmetic `torch.sum` and `torch.mul` operators to shape operators like `torch.reshape` and `torch.transpose`. It also supports more complex matrix operators such as `torch.nn.Linear`, but some of these operators only support combining ciphertext with unencrypted plaintext, limiting their usability in an FHE application. In addition, Concrete ML only has a TFHE implementation, and Concrete ML does not implement floating point operations, so floating point inputs have to be quantized before being used. Quantization allows for the conversion of floats into integers, and lowers the bit width of the input, allowing for higher performance. However, it also necessarily lowers the resolution of the data, increasing the error of the computation. Additionally, all intermediate operations in the circuit must stay under a maximum bit width of 16 bits in order to avoid overflowing the underlying data types, which can mean requiring further quantization to handle operations which increase the bit width of the output. To aid in this, Concrete ML runs a pass through a model with provided sample inputs in a calibration step. This calibration pass serves both to ensure the bit width limit is not exceeded and to tune a quantizer in order to represent the most relevant parts of the input space with higher resolution, decreasing error generated by quantization.

One other limitation of the TFHE scheme, as well as all numerical FHE schemes, is that the polynomial implementation of the ciphertexts is not conducive to running non-linear operations. One solution to non-linear operations in FHE is to find a linear operation that directly approximates the non-linear one. This must be done by the user and can be done in many ways such as creating a Taylor series approximation of the operation[25]. Concrete ML provides another solution in the form of lookup tables compiled for non-linear operations. During calibration, the inputs and outputs of non-linear operations are recorded in a table, which allows for approximation of the function when running it in FHE. This is

used to implement operations such as activation functions in FHE. However, the compute and memory overhead for FHE table lookups can be impractical, especially for large input spaces.

Chapter 3

Motivations

The goal of combining FHE with LLMs is to create a solution where a user can be assured that their query is kept private in a client-server model. Other solutions require the user to trust that a server is not compromised and that an LLM provider does not covertly exfiltrate query data while computing it. FHE, despite its performance and applicability limitations, allows a user to offload intensive operations to a server while not trusting it with the plaintext query in any form. This opens up LLM applications that can run in low compute environments with private information. An example of this would be using LLMs in a medical context. Patient data protection regulations such as the Health Insurance Portability and Accountability Act (HIPAA) can require patient data to be unreadable by third parties, which hamstrings the implementation of client-server LLM interfaces that can make inferences with patient data. An FHE model can allow for patient data to be used without ever revealing its contents to anyone but the key holder. FHE models can also be used to aid in privacy preserving LLM applications in low compute environments. A phone, for example, may not have the resources to run LLM computations locally, but can encrypt and decrypt vectors sent and received from an FHE inference server.

3.1 Performance

FHE performance is generally very slow, on the order of tens to thousands of times slower than the equivalent plaintext operation. Solutions are in development for this, such as FHE hardware accelerators like BASALISC[6], a hardware accelerator concept for the BGV scheme. However, FHE hardware accelerators are not widely adopted, making their usage less practical compared to software based optimizations. Therefore, this project focuses on performance possible with current day general hardware.

Chapter 4

Problem Definition

4.1 Threat Model

The project's threat model involves two parties: the client and the server.

Client

Users interact with the LLM on a client device on which they enter their input and expect their output. The client is assumed to not be compromised, and is trusted by the user. However, only the client should be able to view private information such as the input or the output. This includes any information that could be used to derive the plaintext input or the output, such as the intermediate vectors created when performing an inference.

Server

The client is able to interact with a server through a secure channel in a computationally rich environment in order to offload intensive LLM operations. The server is assumed to have knowledge of the protocol, access to the model's pre-trained weights, and any data sent to it by the client in order to perform LLM inferences. However, the server is controlled by an untrusted party and may be malicious. The server is assumed to able to act outside the bounds of the protocol and manipulate its outputs in order to derive private information from the data is it sent. Therefore, the server cannot be sent any information that could be used to derive the plaintext input or output, and the protocol itself must be able to provide these guarantees even if the server sends manipulated output to the client.

Goals

The goal of this project is to design a system where the threat actor is not able to glean any private information from the user. Private information includes the input contents and the output. This means that any information sent to the server cannot be used by the server

to derive this information. This will be done by only sending the server FHE ciphertexts of information that could be used to derive the input, intermediate products, or the output. However, it is not a goal of the project to protect the pre-trained weights or information about the model protocol itself, as these are considered to be public information known to all parties. Additionally, it is not a goal for the protocol to guarantee correctness or availability when it is under attack. As such, a threat actor is allowed to corrupt the model outputs or deny service to the user as long as this does not lead to the disclosure of private information. It is also not a goal of the project to hide potentially sensitive information unrelated to the inference itself, such the IP address of the client or the frequency of requests. The length of the input itself is also not protected by the system.

Chapter 5

Methodology

5.1 FHE Scheme Selection

CKKS

As detailed in Chapter 2, many different FHE schemes can be employed for the design of this project. The LLaMA model works with floating point operations, so the project initially employed the use of CKKS (a floating point FHE scheme) to approximate the model. This was done using the TenSEAL library mentioned in Section 2.2. Some initial success was made in successfully implementing LLaMA's apply_rotary_embeddings function into the model, which encodes positional data into the initial embeddings. This was possible because apply_rotary_embeddings was composed of simple vector additions and multiplications, and operated on relatively low size inputs. In testing, the encryption of inputs took 6.04 seconds, the decryption of inputs took 10.17 seconds, and the operation itself took 21.64 seconds on average.

However, approaching the problem with CKKS ended up being impractical due to its slow performance. The smallest release of LLaMA 2 features seven billion parameters, and the linear transformation operations in its attention and feed forward operations can work with matrices up to 4096 by 11008 elements. Attempting to implement these using TenSEAL was impossible, as the library would attempt to create one CPU thread for each element in the result vector, causing resource thrashing on the test machine. However, a smaller matrix multiplication of two 10 by 10 matrices took 0.8 seconds on average, with a larger 15x15 matrix operation taking 4.9 seconds. Even with an overly optimistic assumption that matrices have a time complexity of $O(n^2)$ relative to their input size, extrapolating this to a 4096 by 11008 matrix leads to extremely high estimated times measuring in the millions of days for the operation to complete. However, in reality, this would take even longer than the estimate. This is because of the limited multiplicative depth of a CKKS ciphertext. The multiplicative depth of a ciphertext can be increased by adding more polynomial moduli to it, but this comes at a significant performance cost. Alternatively, the ciphertext can be sent back to the client, decrypted, and re-encrypted to reset its multiplicative depth, but

each of these rounds of communication would take many seconds to encrypt and decrypt the ciphertext. These limitations make it even more impractical to use CKKS.

TFHE

The Concrete ML library described in section 2.2 was used to integrate a TFHE based scheme into the model. TFHE has significant performance gains over CKKS due to the increased efficiency of arithmetic operations on the ciphertext structure and the increased speed of lower precision arithmetic. When running the same linear transformation operations mentioned above, a 4096x11008 matrix multiplication can be done in 70-100 seconds using 16 bit precision. These metrics make TFHE very suited to handle the bulk of the model's operations. However, it is important to note that TFHE dramatically increases ciphertext sizes compared to CKKS. In addition, it is an integer based scheme and using it in the model requires quantizing the inputs.

Conclusion: Using the TFHE scheme allows for doing intensive computations in a feasible amount of time. However, it is an integer based scheme and requires quantization in order to use in LLaMA, which has floating point operations.

5.2 Quantization

As detailed in section 2.2, Concrete ML supports the quantization of inputs during compilation, where the sample inputs given during compilation help to calibrate a quantizer for the circuit. In order to determine the best quantization, the model was run with several samples to determine how best to quantize the inputs with the highest granularity possible for a given bit width. These inputs were generated using the Datasets[15] library from HuggingFace which provides many different datasets for ML purposes. Specifically, the calibration process was done on random samples from the Wikipedia dataset in to get a good sample of natural language. The calibrated quantizer is then included with the compiled model so that the client has access to it when generating ciphertexts from the floating point vectors. Once an output is received from the server, the ciphertext is then reconstructed into its associated floating point tensor, allowing the client to continue computing on the output tensor using floating point operations. This serves to minimize the error propagated from quantization by only using it where it is needed. The bit width of the quantization is important, as a larger bit width means less error while a smaller one improves performance both in lowering the amount of computational intensity as well as bandwidth usage when transmitting the ciphertext. After some trials, it was determined that 16 bits would be the best performance-accuracy trade-off for the hybrid model (see section 6.5 for detailed results).

Conclusion: The calibrated quantization used by the Concrete ML library helps achieve the best accuracy for operator quantization. Maximizing the quantization bit width is important for minimizing error.

5.3 Operation Selection

The selection of operations to do in FHE is a critical design choice. FHE has varying levels of compatibility and performance for each operation, and some low computation operations may be trivial for the client to do in plaintext, but take significantly more resources for the server to do in FHE. Thus, the following section is an investigation of LLaMA operations and how well they perform when done in TFHE.

Linear Transformation

In LLaMA, linear transformations are used extensively in both the self attention and feed forward mechanisms. In self attention, they are used to project the input into query, key, and value vectors, as well as to project the attention values back into an output. In the feed forward operation, linear transformations are used to transform the input and output for the feed forward layers. All of these operations involve a large matrix multiplication of inputs with the pre-trained weights. For a seven billion parameter implementation of LLaMA2, each token of output requires 32 layers of seven different linear transformations that multiply their inputs by a cumulative total of 7 billion parameters total per token. Loading these parameters into memory is what is responsible for the vast majority of LLaMA's memory requirement, and being able to offload these to a server would decrease the clientside computational intensity and memory requirements of the model greatly. Fortunately, as seen in the benchmarks done previously, TFHE is well equipped to do large matrix multiplications. Additionally, the pre-trained weights themselves can be used in plaintext, as they are not a private part of the model. This allows for even better performance and lower memory usage by the server. The final implementation features all linear transformations in the model done serverside in FHE, greatly decreasing the compute and memory needs for the client (see section 6.1 for memory usage in detail).

Attention KV Cache and Weighting

LLaMA incorporates linear transformations in its `Attention` block, which then stores the calculated attention weights in a key value cache in order to pre-compute all the attention weights at once and save on duplicated work for each token. While the key value projections are done in FHE on the server, the rest of the scoring is done clientside, as scoring involves much fewer computationally intensive operations (some fairly small matrix multiplications and a softmax). Additionally, the sizes of the ciphertexts mean that it would take much more memory to have the cache on the server compared to on the client. Therefore, it makes the most sense to keep the KV cache and the rest of the scoring on the client, as it saves on server memory usage and allows the client to perform some lighter computations. While computing operations in plaintext means that parts of the protocol could be readable to a threat actor, this is acceptable because these operations are done purely on the client,

meaning that the threat actor is unable to read the plaintext and only receives ciphertext from the client for the linear transformations to be computed over FHE.

Non-Linear Operations

One of the core operations in LLaMA is the SwiGLU activation function in the feed forward part of the model. It serves the purpose of introducing non-linearity into the model computations. This portion of the model, as well as other non linear operations such as softmax and RMSNorm, are very difficult to do in FHE. SwiGLU is non-monotonic, making it difficult to find an accurate linear approximation of the operation. Concrete ML provides table lookups to approximate the operator, but using them dramatically increases the error of the function's output, causing the final output to be garbled. As such, SwiGLU, alongside other non-linear functions such as RMSNorm, are computed clientside. These operators are acceptable to run clientside because of their low computational intensity. Activation and normalization functions have a linear runtime complexity with respect to the size of their inputs, which means that they can still be computed quickly in a low compute environment such as the client.

Conclusion: Offloading the linear transformation operations to do in FHE on the server provides a good trade-off of FHE performance and client computational intensity. It runs fairly quickly on the server and removes a significant amount of the memory requirement for the client (see 6.1 for details). Keeping non-linear operations and the KV cache clientside allows the client to perform some low intensity operations that do not require a large amount of memory, such as activation functions with a linear runtime.

5.4 Final Model

Below is a diagram of the final model, where the client is a LLaMA2 model augmented with remote FHE calls for the linear transforms. To offload these calls to the server, the client quantizes and encrypts its inputs before sending them and the compiled FHE circuit to the server. The server implements a REST API which accepts these calls and computes the circuit over the ciphertext, sending the output ciphertext back to the server. The client then decrypts these ciphertexts back into floats for local computation. This hybrid client-server model offloads the most computationally intensive parts of the inference while always preserving the privacy of the inputs, intermediate vectors, and outputs.

The names of the modules that are computed in FHE on the server are as follows:

1. `Attention.wq`
2. `Attention.wk`
3. `Attention.wv`

4. `Attention.wo`
5. `FeedForward.w1`
6. `FeedForward.w2`
7. `FeedForward.w3`

The operations and modules that are computed in plaintext on the client are as follows:

1. `Transformer.tok_embeddings`
2. `Transformer.norm`
3. `Transformer.output`
4. `TransformerBlock.attention_norm`
5. `TransformerBlock.ffn_norm`
6. The SwiGLU operator and the matrix multiplication within the FeedForward block.
7. The KV caches within the Attention block.
8. The transpose, softmax, and matmul operator used to calculate the scores within the Attention block.

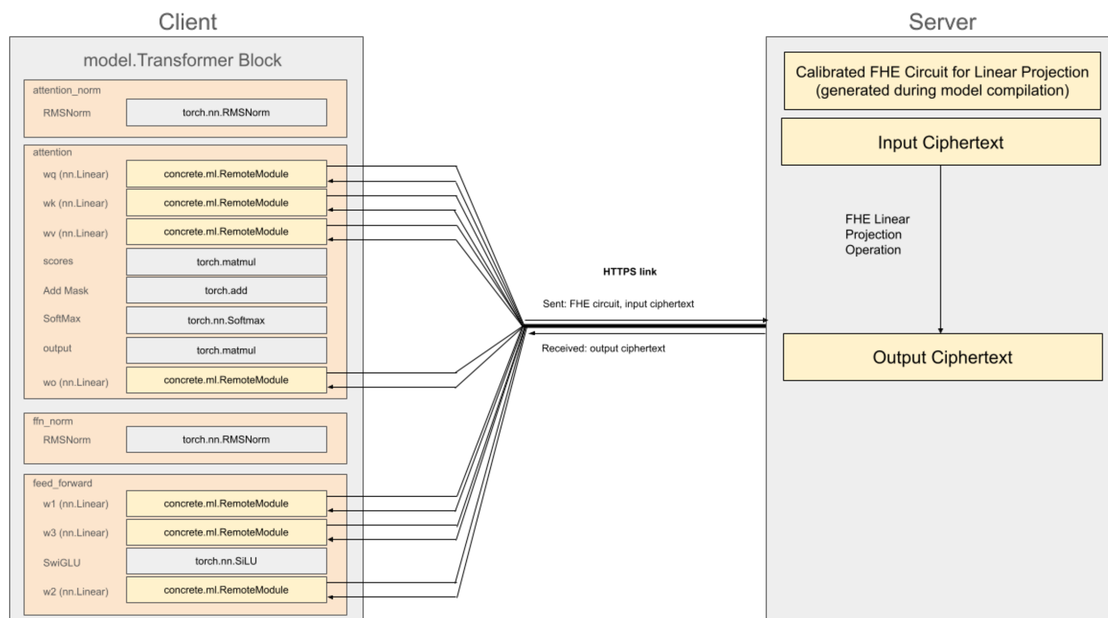


Figure 5.1: A diagram of the hybrid client-server model

Chapter 6

Results

Experimental Questions

This chapter focuses on finding results for the following experimental questions. These questions provide context for measurements taken while experimenting with the model's design, and illustrate how the model's design affects its performance and practicality.

1. How much memory is saved by the client when using the hybrid model compared to running the model on the client? How much memory is required by the server to complete an inference?
2. How much network bandwidth is used for the inference operation?
3. How much time does the inference operation take?
4. How does changing the size of the input change the runtime and the network bandwidth?
5. How does changing the quantized bit width of the FHE operation change the accuracy, runtime, and network bandwidth of the model?
6. How does adding more processors change the runtime of the inference operation?

Data Collection

Results were collected on a machine with an AMD EPYC 9654 96-Core Processor alongside 1.5 Tb of memory. While GPU acceleration is possible on the client side, the client does not use GPU acceleration, as the encryption and decryption happens on the CPU, meaning the time spent transferring data to and from the GPU was longer than the time saved running low intensity operations on the GPU. Unless otherwise specified, the `FHEModelServer` was run on 12 cores. The client ran on the same machine and they communicated on a local network socket in order to maximize bandwidth. All results were collected for the LLaMA2

model with seven billion parameters. Unless specified, the model is instructed to complete the 7 token input "Computations over encrypted data can help" with a temperature of 0.8, top_p of 0.95, and FHE portions of the model quantized to 16 bits.

6.1 Memory Usage

Because the pretrained weights for the linear transformations no longer needed to be on the client, the model's memory usage was lowered to approximately 2 gigabytes on the clientside for the model itself. While the client also had to store the compiled FHE model data (approximately 40 gigabytes) to transmit to the `FHEModelServer`, in a production deployment of this project the server could cache the compiled models knowing they would always be the same for each query, meaning the client would not need to load them into memory to send to the server. Each token has to be stored in the KV cache, but the amount stored per token is negligible at less than ten megabytes per token. Thus, the client's memory usage goes down from approximately 14 gigabytes of memory to 2 gigabytes when removing the compiled FHE model data.

The maximum amount of memory used by the FHE server was 201 gigabytes. This means that for every gigabyte of memory saved by the client, the server uses approximately 16.5 gigabytes. This significant increase in total memory usage is likely related to the larger sizes of ciphertext vectors, as well as the need for large keys for programmable bootstrapping and relinearization operations.

6.2 Bandwidth Usage

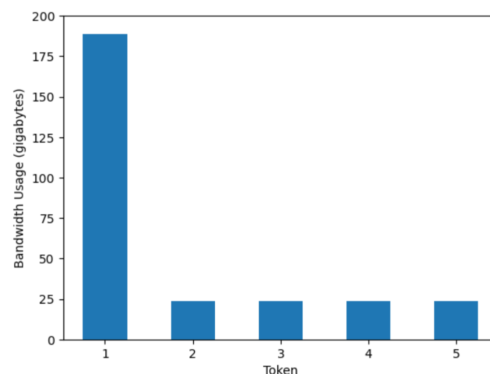


Figure 6.1: Bandwidth per token on a five token output from a seven token input

For a five token output from a seven token input, the model used approximately 283 gigabytes of bandwidth, the vast majority of which was used to download the ciphertext back to the client from the server. The first token required around twice the bandwidth as the remaining tokens combined due to the increased size of the inputs for self attention.

6.3 Token Inference Speed

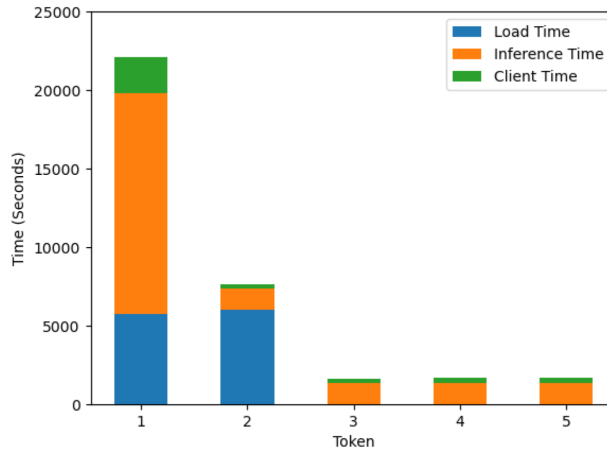


Figure 6.2: Time per token on a five token output from a seven token input.

- Load time is the amount of time the `FHEModelServer` spends to load the MLIR circuit generated by the compiler and sent to the server.
- Inference time is the time actually spent running FHE linear projections on the ciphertext.
- Client time encompasses time spent by the client to run the clientside part of the model, as well as encrypt and decrypt the ciphertext.

For a five token output from a seven token input, the model took 34,631 seconds to generate an output. Of this time, the vast majority was spent on the first two tokens waiting for the server to load the FHE circuits needed for computing the linear transformations. For example, the attention.wo transform needs to load a circuit to accommodate an input with shape `(1, 8, 4096)` for the first token. For the second and subsequent tokens, it loads a circuit to accommodate an input with shape `(1, 1, 4096)`. This results in no load time needed for subsequent tokens as the circuit is already loaded from the previous iteration.

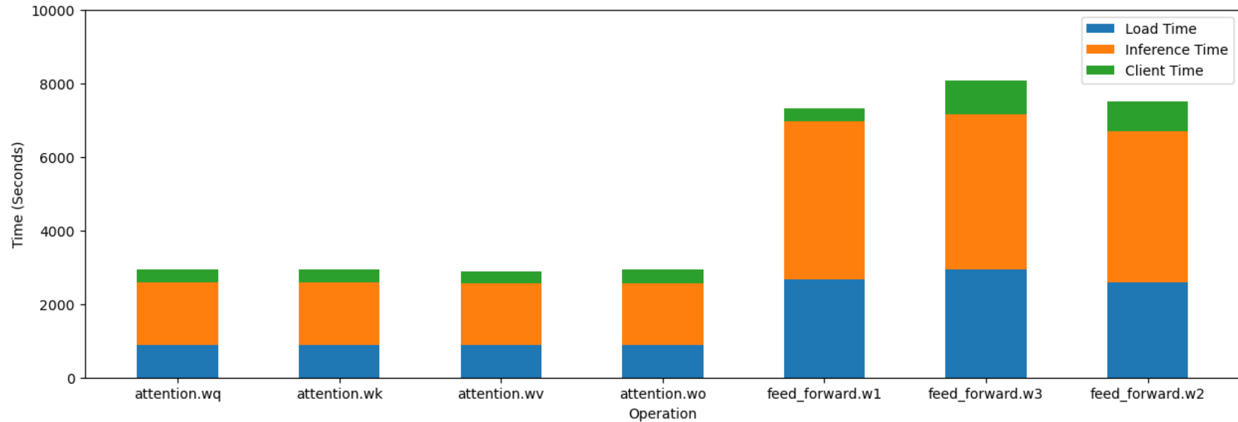


Figure 6.3: Total time spent on each FHE operator

When breaking down the total time spent per FHE operation, each attention operator takes around 8.5% of the total time. The the feed forward operations take between 21% and 23% of the total time each. The time spent on each operator itself scales roughly linearly with the size of the input, with the 11008 element feed-forward linear transformations taking roughly three times as long to run as the 4096 element linear attention transformations.

6.4 Input Size Impact

The size of the input has a significant impact on model speed, much more than the output size. This is consistent with the findings above where tokens after the first and second token have a greatly diminished runtime due to not needing to load a new circuit. However, increasing the input increases the size of the first ciphertexts that are run by the FHEModelServer, both increasing bandwidth and inference time. Bandwidth scales from 137 gigabytes with a one token input to 283 gigabytes with an seven token input. Inference time scales similarly from 21,804 to 34,630 seconds.

6.5 Quantization Impact

Accuracy

Changing the bit width of the ciphertexts has a significant impact on the accuracy of the output. This was measured by compiling the model for each bit width on the same calibration input and getting the output logits for the first token of the output. Below shows the mean squared error (MSE) when comparing these logits to the output logits of an unmodified LLaMA2-7b model, with no hybrid FHE model. The MSE metric allows us to quantify how different the logits are in a way that emphasizes the impact of larger differences between

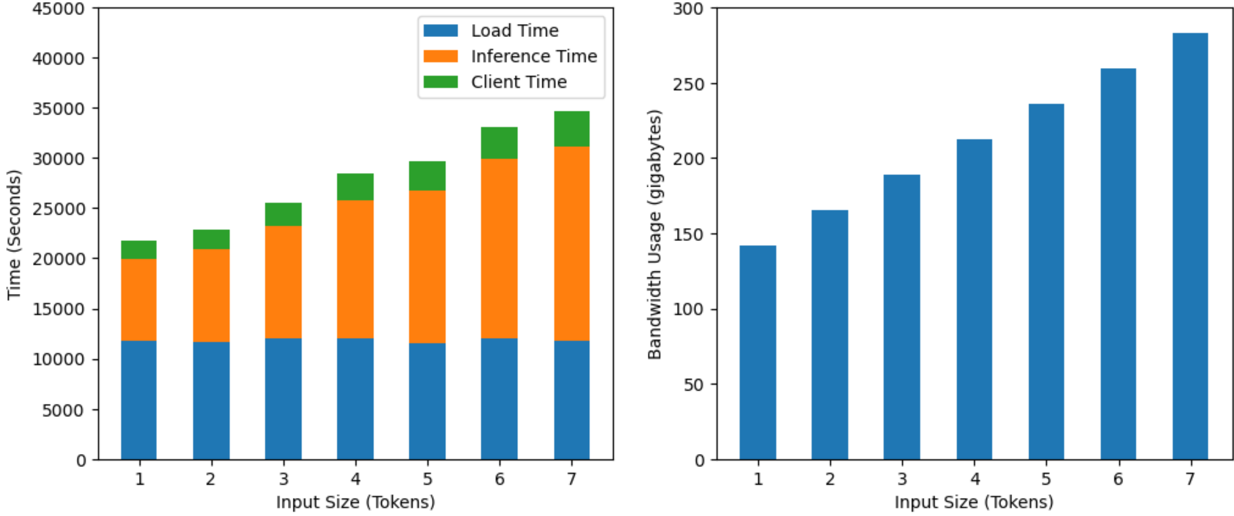


Figure 6.4: Total time and bandwidth spent on a five token output based on the size of the input, scaling from one to seven tokens

individual logits. This is important because large differences between individual logits increases the chance of the hybrid model selecting a different token than the unmodified model would have selected, which impacts the accuracy of the output as observed by the user.

Bit Width	MSE
16	0.0068
14	0.0243
12	0.4754
10	0.2688
8	1.2974
6	137.0635

Table 6.1: MSE of first token logits of models with various bit widths for quantized ciphertext.

The error grows exponentially as the bit width shrinks, reflecting the expected exponential error growth inherent of noisy inputs to a feedback loop. Because error in the logits affects the selection of the next token, error in selecting one token led to increased error when computing subsequent tokens, meaning that even a 14 bit quantized model had a significantly lower quality than a 16 bit model.

Performance

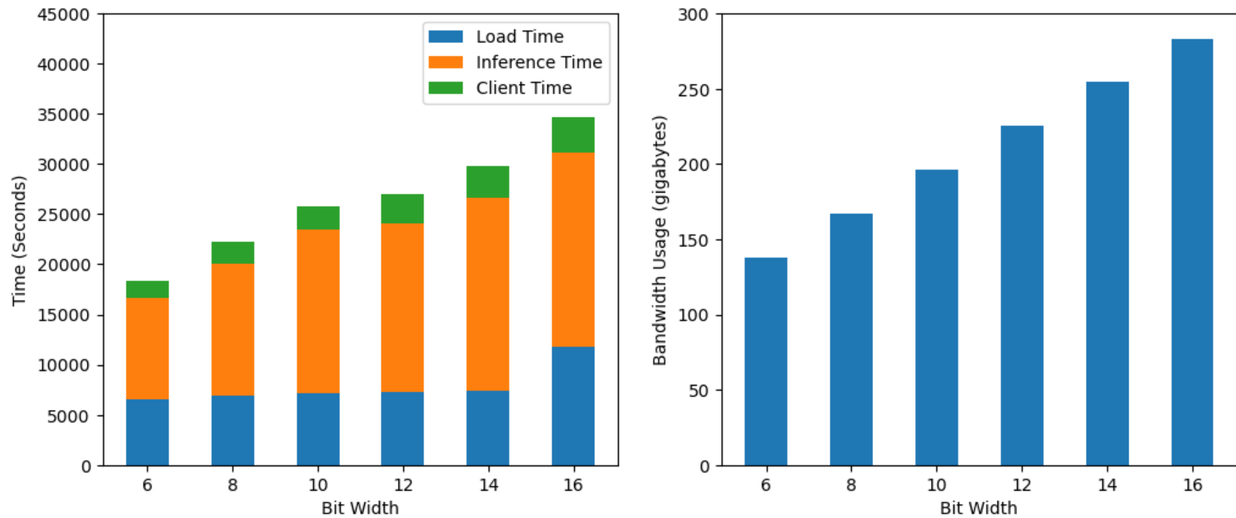


Figure 6.5: Total time and bandwidth spent on a five token output from a seven token input based on the quantization of the model

The bit width of the ciphertexts had an impact on both total bandwidth usage and inference speeds on the server. While load times were unaffected (except for the 16 bit width, possibly due to the limitations in serializing 16 bit ciphertexts), the inference time scaled linearly based on the bit width of the encrypted inputs. The bandwidth also scaled linearly with the bit width.

6.6 Parallelization Impact

While parallelization had no impact on the circuit load times for the `FHEModelServer`, it had a significant impact on inference speed itself, with total inference time going from 19,394 seconds with 12 processors to 5,504 seconds with 192 processors. The impact of adding more processors has diminishing returns, and after 96 processors it generally does not make sense to continue adding any more. This is likely because of Amdahl's law[7], which states that the parallelizable part of a task has an exponentially decreasing marginal speedup for each processor added to the task. FHE inference is likely composed of inherently sequential tasks which do not take advantage of additional processors, as well as a parallelizable tasks which can take advantage of the processors, but only to a certain extent before the marginal speedup is negligible.

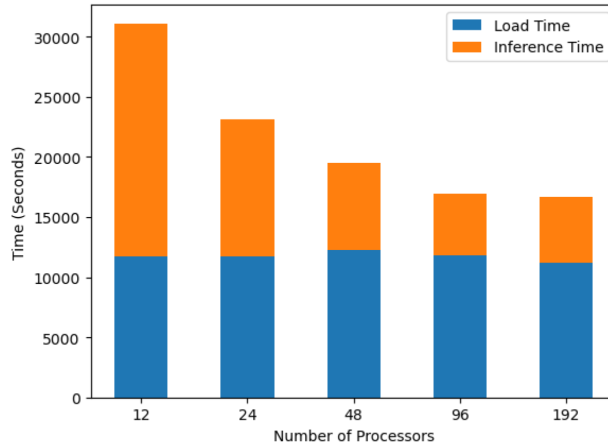


Figure 6.6: Total load and inference time spent on a five token output from a seven token input based on the number of processors assigned to the server

6.7 Conclusions

The answers to the experimental questions at the start of the section are as follows:

1. The client saves 12 gigabytes of memory usage compared to regular inference. There is additional memory usage to store the compiled FHE model data, but it is trivially possible to remove this by caching the model on the server. The server requires 201 gigabytes to complete an inference, meaning there is approximately a 16.5:1 ratio between memory used by the server and memory saved by the client.

2. For the default experimental parameters, the model used approximately 283 gigabytes of bandwidth, with the majority being used for the first token self attention.

3. For the default experimental parameters, the model took approximately 9.6 hours to generate an output with the experimental parameters. The majority of this time was spent on the first and second tokens to load the compiled models and run self attention.

4. The runtime and bandwidth requirements scale roughly linearly with the size of the input.

5. The runtime and bandwidth requirements scale roughly linearly with the bit-width of the FHE inputs. The error scales exponentially as the bit width shrinks, with noticeable loss of output quality happening for any bit width under 16 bits.

6. Adding more processors has no impact on the time spent loading the FHE circuits, but does speed up the inference. However, this speedup has diminishing returns and stops being significant after 96 total processors.

6.8 Future Work

This model has significant opportunities for improvement in order to reach practicality and applicability in a wide variety of use cases.

Circuit Load Optimizations

A lot of duplicate work is being done in this model to load duplicate circuits, as the underlying server implementation does not optimize for the fact that the same circuits are used for layers 0 through 31, and reloads the same circuits for each one. De-duplicating that work by reusing the same circuit each time would lead to a potential 32x optimization of load times. In addition, a production deployment of this model could pre-load the FHE circuits needed for various inputs, optimizing out the load time entirely.

Fully Serverside FHE

One potential opportunity is to make a fully end to end FHE model, rather than a hybrid one. This would require being able to create linear approximations of the activation functions and increase work needed serverside, but would mean that the client only has to encrypt the tokenized input and decrypt the output received from the server, greatly decreasing clientside computational intensity. End-to-end FHE would also decrease the bandwidth required, as only one round of communication would be needed between the client and the server. However, reaching this ideal would require significant improvements to FHE performance and more methods of handling error accumulation.

Different FHE Schemes

A way to improve performance in the hybrid model would be to use different FHE schemes based on what is the most performant for a given module. This could include changing the quantization bit width on a block by block basis rather than setting them all to sixteen bits. It could also involve potentially using other protocols entirely such as BGV or CKKS if they would be more practical for a given block.

Different Models

Some LLMs may be more conducive to being implemented in FHE. This would depend on how many parameters they have as well as the nature of their activation functions and other non-linear operations.

Hardware Acceleration

An FHE server with hardware accelerators may be able to do computations over ciphertext with greater speed and accuracy. Some implementations of FHE accelerators require custom

hardware but others can run on typical datacenter GPUs and take advantage of their large vectorized operations[20]. Integrating these with the hybrid model may increase inference speeds.

Network Optimization

Practical uses of hybrid models require significant amounts of bandwidth in order to transfer ciphertexts between the client and server. Finding ways of compressing ciphertexts or caching ones that may repeat would increase the speed of the model, especially in low bandwidth environments.

Bibliography

[1] Ayoub Benaissa et al. *TenSEAL: A Library for Encrypted Tensor Operations Using Homomorphic Encryption*. 2021. arXiv: 2104.03152 [cs.CR]. URL: https://arxiv.org/abs/2104.03152.

[2] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. "(Leveled) fully homomorphic encryption without bootstrapping". In: *Proceedings of the 3rd Innovations in Theoretical Computer Science Conference*. ITCS '12. Cambridge, Massachusetts: Association for Computing Machinery, 2012, pp. 309–325. ISBN: 9781450311151. DOI: 10.1145/2090236.2090262. URL: https://doi.org/10.1145/2090236.2090262.

[3] Jung Hee Cheon et al. *Homomorphic Encryption for Arithmetic of Approximate Numbers*. Cryptology ePrint Archive, Paper 2016/421. https://eprint.iacr.org/2016/421. 2016. URL: https://eprint.iacr.org/2016/421.

[4] Ilaria Chillotti et al. *TFHE: Fast Fully Homomorphic Encryption Library*. https://tfhe.github.io/tfhe/ August 2016.

[5] Siddharth Dongre et al. "Quantifying the Costs of Data Breaches". In: *Critical Infrastructure Protection XIII*. Ed. by Jason Staggs and Sujeet Shenoi. Cham: Springer International Publishing, 2019, pp. 3–16. ISBN: 978-3-030-34647-8.

[6] Robin Geelen et al. *BASALISC: Programmable Hardware Accelerator for BGV Fully Homomorphic Encryption*. 2023. arXiv: 2205.14017 [cs.CR]. URL: https://arxiv.org/abs/2205.14017.

[7] John L. Gustafson. "Amdahl's Law". In: *Encyclopedia of Parallel Computing*. Ed. by David Padua. Boston, MA: Springer US, 2011, pp. 53–60. ISBN: 978-0-387-09766-4. DOI: 10.1007/978-0-387-09766-4_77. URL: https://doi.org/10.1007/978-0-387-09766-4_77.

[8] Shai Halevi and Victor Shoup. *Design and implementation of HElib: a homomorphic encryption library*. Cryptology ePrint Archive, Paper 2020/1481. 2020. URL: https://eprint.iacr.org/2020/1481.

[9] Meng Hao et al. "Iron: Private Inference on Transformers". In: *Advances in Neural Information Processing Systems*. Ed. by S. Koyejo et al. Vol. 35. Curran Associates, Inc., 2022, pp. 15718–15731. URL: https://proceedings.neurips.cc/paper_files/paper/2022/file/64e2449d74f84e5b1a5c96ba7b3d308e-Paper-Conference.pdf.

[10] Chien-Lung Hsu et al. "Privacy Concerns and Information Sharing: The Perspective of the U-Shaped Curve". In: *Frontiers in Psychology* 13 (2022). ISSN: 1664-1078. DOI: 10.3389/fpsyg.2022.771278. URL: https://www.frontiersin.org/journals/psychology/articles/10.3389/fpsyg.2022.771278.

[11] Ameya D. Jagtap and George Em Karniadakis. *How important are activation functions in regression and classification? A survey, performance comparison, and future directions*. 2022. arXiv: 2209.02681 [cs.LG]. URL: https://arxiv.org/abs/2209.02681.

[12] Wonkyung Jung et al. "Accelerating Fully Homomorphic Encryption Through Architecture-Centric Analysis and Optimization". In: *IEEE Access* 9 (2021), pp. 98772–98789. ISSN: 2169-3536. DOI: 10.1109/access.2021.3096189. URL: http://dx.doi.org/10.1109/ACCESS.2021.3096189.

[13] *Lattigo v5*. Online: https://github.com/tuneinsight/lattigo. EPFL-LDS, Tune Insight SA. Nov. 2023.

[14] Dayeol Lee et al. "Keystone: an open framework for architecting trusted execution environments". In: *Proceedings of the Fifteenth European Conference on Computer Systems*. EuroSys '20. Heraklion, Greece: Association for Computing Machinery, 2020. ISBN: 9781450368827. DOI: 10.1145/3342195.3387532. URL: https://doi.org/10.1145/3342195.3387532.

[15] Quentin Lhoest et al. *Datasets: A Community Library for Natural Language Processing*. 2021. arXiv: 2109.02846 [cs.CL]. URL: https://arxiv.org/abs/2109.02846.

[16] Yehuda Lindell. "Secure multiparty computation". In: *Commun. ACM* 64.1 (Dec. 2020), pp. 86–96. ISSN: 0001-0782. DOI: 10.1145/3387108. URL: https://doi.org/10.1145/3387108.

[17] Wen-jie Lu et al. *BumbleBee: Secure Two-party Inference Framework for Large Transformers*. Cryptology ePrint Archive, Paper 2023/1678. 2023. URL: https://eprint.iacr.org/2023/1678.

[18] Jinglong Luo et al. *SecFormer: Towards Fast and Accurate Privacy-Preserving Inference for Large Language Models*. 2024. arXiv: 2401.00793 [cs.LG]. URL: https://arxiv.org/abs/2401.00793.

[19] Chiara Marcolla et al. *Survey on Fully Homomorphic Encryption, Theory, and Applications*. Cryptology ePrint Archive, Paper 2022/1602. https://eprint.iacr.org/2022/1602. 2022. DOI: 10.1109/JPROC.2022.3205665. URL: https://eprint.iacr.org/2022/1602.

[20] Shintaro Narisada et al. *GPU Acceleration of High-Precision Homomorphic Computation Utilizing Redundant Representation*. Cryptology ePrint Archive, Paper 2023/1467. https://eprint.iacr.org/2023/1467. 2023. DOI: 10.1145/3605759.3625256. URL: https://eprint.iacr.org/2023/1467.

[21] Humza Naveed et al. *A Comprehensive Overview of Large Language Models*. 2024. arXiv: 2307.06435 [cs.CL]. URL: https://arxiv.org/abs/2307.06435.

[22] Qi Pang et al. "BOLT: Privacy-Preserving, Accurate and Efficient Inference for Transformers". In: *2024 IEEE Symposium on Security and Privacy (SP)*. 2024, pp. 4753–4771. DOI: 10.1109/SP54263.2024.00130.

[23] Iqbal H. Sarker. "LLM potentiality and awareness: a position paper from the perspective of trustworthy and responsible AI modeling". In: *Discover Artificial Intelligence* 4.1 (May 2024), p. 40. ISSN: 2731-0809. DOI: 10.1007/s44163-024-00129-0. URL: https://doi.org/10.1007/s44163-024-00129-0.

[24] *Microsoft SEAL (release 4.1)*. https://github.com/Microsoft/SEAL. Microsoft Research, Redmond, WA. Jan. 2023.

[25] Zvonimir Šikić. "Taylor's theorem". In: *International Journal of Mathematical Education in Science and Technology* 21 (Mar. 1990), pp. 111–115. DOI: 10.1080/0020739900210115.

[26] Ashish Vaswani et al. *Attention Is All You Need*. 2023. arXiv: 1706.03762 [cs.CL]. URL: https://arxiv.org/abs/1706.03762.

[27] Febrianti Wibawa et al. "BFV-Based Homomorphic Encryption for Privacy-Preserving CNN Models". In: *Cryptography* 6.3 (2022). ISSN: 2410-387X. DOI: 10.3390/cryptography6030034. URL: https://www.mdpi.com/2410-387X/6/3/34.

[28] Zama. *Concrete ML: a Privacy-Preserving Machine Learning Library using Fully Homomorphic Encryption for Data Scientists.* https://github.com/zama-ai/concrete-ml. 2022.

[29] Zama. *Concrete: TFHE Compiler that converts python programs into FHE equivalent.* https://github.com/zama-ai/concrete. 2022.

[30] Jiawen Zhang et al. *Secure Transformer Inference Made Non-interactive*. Cryptology ePrint Archive, Paper 2024/136. 2024. URL: https://eprint.iacr.org/2024/136.